目录

TypeScript 入门教程

从 JavaScript 程序员的角度总结思考，循序渐进的理解 TypeScript。

关于本书

- 在线阅读（部署在 GitBook 上，可能需要翻墙）
- 在线阅读（GitHub 版）
- GitHub 地址
- 作者：xcatliu
- 官方群：加入QQ群 767142358

本书是作者在学习 TypeScript 后整理的学习笔记。

随着对 TypeScript 理解的加深和 TypeScript 社区的发展，本书也会做出相应的更新，欢迎大家 Star 收藏。

- 发现文章内容有问题，可以直接在页面下方评论
- 对项目的建议，可以提交 issue 向作者反馈
- 欢迎直接提交 pull-request 参与贡献

为什么要写本书

TypeScript 虽然有官方手册及其非官方中文版，但是它每一章都希望能详尽的描述一个概念，导致前面的章节就会包含很多后面才会学习到的内容，而有些本该一开始就了解的基础知识却在后面才会涉及。如果是初学者，可能需要阅读多次才能理解。所以它更适合用来查阅，而不是学习。

与官方手册不同，本书着重于从 JavaScript 程序员的角度总结思考，循序渐进的理解 TypeScript，希望能给大家一些帮助和启示。

由于一些知识点与官方手册重合度很高，本书会在相应章节推荐直接阅读中文手册。

关于 TypeScript

TypeScript 是 JavaScript 的一个超集，主要提供了类型系统和对 **ES6** 的支持，它由 Microsoft 开发，代码开源于 GitHub 上。

它的第一个版本发布于 2012 年 10 月，经历了多次更新后，现在已成为前端社区中不可忽视的力量，不仅在 Microsoft 内部得到广泛运用，而且 Google 的 Angular2 也使用了 TypeScript 作为开发语言。

适合人群

本书适合以下人群

- 熟悉 JavaScript，至少阅读过一遍《JavaScript 高级程序设计》
- 了解 ES6，推荐阅读 ECMAScript 6 入门
- 了解 Node.js，会用 npm 安装及使用一些工具
- 想了解 TypeScript 或者想对 TypeScript 有更深的理解

本书不适合以下人群

- 没有系统学习过 JavaScript
- 已经能够很熟练的运用 TypeScript

评价

> 《TypeScript 入门教程》全面介绍了 TS 强大的类型系统，完整而简洁，示例丰富，比官方文档更易读，非常适合作为初学者学习 TS 的第一本书。
>
> —— 阮一峰

目录

版权许可

相关资料

- TypeScript 官网
- Handbook（中文版）
- ECMAScript 6 入门

- 下一章：简介

简介

本部分介绍了在学习 TypeScript 之前需要了解的知识，具体内容包括：

- 什么是 TypeScript
- 安装 TypeScript
- Hello TypeScript

- 上一章：前言
- 下一章：什么是 TypeScript

什么是 TypeScript

首先，我对 TypeScript 的理解如下：

TypeScript 是 JavaScript 的一个超集，主要提供了类型系统和对 **ES6** 的支持，它由 Microsoft 开发，代码开源于 GitHub 上。

其次引用官网的定义：

> TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Any browser. Any host. Any OS. Open source.

翻译成中文即是：

> TypeScript 是 JavaScript 的类型的超集，它可以编译成纯 JavaScript。编译出来的 JavaScript 可以运行在任何浏览器上。TypeScript 编译工具可以运行在任何服务器和任何系统上。TypeScript 是开源的。

为什么选择 TypeScript

TypeScript 官网列举了一些优势，不过我更愿意自己总结一下：

TypeScript 增加了代码的可读性和可维护性

- 类型系统实际上是最好的文档，大部分的函数看看类型的定义就可以知道如何使用了
- 可以在编译阶段就发现大部分错误，这总比在运行时候出错好
- 增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等

TypeScript 非常包容

- TypeScript 是 JavaScript 的超集，`.js` 文件可以直接重命名为 `.ts` 即可
- 即使不显式的定义类型，也能够自动做出类型推论
- 可以定义从简单到复杂的一切类型
- 即使 TypeScript 编译报错，也可以生成 JavaScript 文件
- 兼容第三方库，即使第三方库不是用 TypeScript 写的，也可以编写单独的类型

文件供 TypeScript 读取

TypeScript 拥有活跃的社区

- 大部分第三方库都有提供给 TypeScript 的类型定义文件
- Google 开发的 Angular2 就是使用 TypeScript 编写的
- ES6 的一部分特性是借鉴的 TypeScript 的（这条需要来源）
- TypeScript 拥抱了 ES6 规范，也支持部分 ES7 草案的规范

TypeScript 的缺点

任何事物都是有两面性的，我认为 TypeScript 的弊端在于：

- 有一定的学习成本，需要理解接口（Interfaces）、泛型（Generics）、类（Classes）、枚举类型（Enums）等前端工程师可能不是很熟悉的东西。而且它的中文资料也不多
- 短期可能会增加一些开发成本，毕竟要多写一些类型的定义，不过对于一个需要长期维护的项目，TypeScript 能够减少其维护成本（这条需要来源）
- 集成到构建流程需要一些工作量
- 可能和一些库结合的不是很完美（这条需要举例）

大家可以根据自己团队和项目的情况判断是否需要使用 TypeScript。

- 上一章：简介
- 下一章：安装 TypeScript

安装 TypeScript

TypeScript 的命令行工具安装方法如下：

```
npm install -g typescript
```

以上命令会在全局环境下安装 `tsc` 命令，安装完成之后，我们就可以在任何地方执行 `tsc` 命令了。

编译一个 TypeScript 文件很简单：

```
tsc hello.ts
```

我们约定使用 TypeScript 编写的文件以 `.ts` 为后缀。

编辑器

TypeScript 最大的优势之一便是增强了编辑器和 IDE 的功能，包括代码补全、接口提示、跳转到定义、重构等。

主流的编辑器都支持 TypeScript，这里我推荐使用 Visual Studio Code。

它是一款开源，跨终端的轻量级编辑器，内置了 TypeScript 支持。

另外它本身也是用 TypeScript 编写的。

下载安装：https://code.visualstudio.com/

获取其他编辑器或 IDE 对 TypeScript 的支持：

- Sublime Text
- Atom
- WebStorm
- Vim
- Emacs
- Eclipse
- Visual Studio 2015

- Visual Studio 2013

- 上一章：什么是 TypeScript
- 下一章：Hello TypeScript

Hello TypeScript

我们从一个简单的例子开始。

将以下代码复制到 `hello.ts` 中：

```
function sayHello(person: string) {
    return 'Hello, ' + person;
}

let user = 'Tom';
console.log(sayHello(user));
```

然后执行

```
tsc hello.ts
```

这时候会生成一个编译好的文件 `hello.js` ：

```
function sayHello(person) {
    return 'Hello, ' + person;
}
var user = 'Tom';
console.log(sayHello(user));
```

TypeScript 中，使用 `:` 指定变量的类型，`:` 的前后有没有空格都可以。

上述例子中，我们用 `:` 指定 `person` 参数类型为 `string` 。但是编译为 js 之后，并没有什么检查的代码被插入进来。

TypeScript 只会进行静态检查，如果发现有错误，编译的时候就会报错。

> `let` 是 ES6 中的关键字，和 `var` 类似，用于定义一个局部变量，可以参阅 let 和 const 命令。

下面尝试把这段代码编译一下：

```
function sayHello(person: string) {
    return 'Hello, ' + person;
}

let user = [0, 1, 2];
console.log(sayHello(user));
```

编辑器中会提示错误，编译的时候也会出错：

```
index.ts(6,22): error TS2345: Argument of type 'number[]' is not
 assignable to parameter of type 'string'.
```

但是还是生成了 js 文件：

```
function sayHello(person) {
    return 'Hello, ' + person;
}
var user = [0, 1, 2];
console.log(sayHello(user));
```

TypeScript 编译的时候即使报错了，还是会生成编译结果，我们仍然可以使用这个编译之后的文件。

如果要在报错的时候终止 js 文件的生成，可以在 `tsconfig.json` 中配置 `noEmitOnError` 即可。关于 `tsconfig.json`，请参阅官方手册（中文版）。

- 上一章：安装 TypeScript
- 下一章：基础

基础

本部分介绍了 TypeScript 中的常用类型和一些基本概念，旨在让大家对 TypeScript 有个初步的理解。具体内容包括：

- 原始数据类型
- 任意值
- 类型推论
- 联合类型
- 对象的类型——接口
- 数组的类型
- 函数的类型
- 类型断言
- 声明文件
- 内置对象

- 上一章：Hello TypeScript
- 下一章：原始数据类型

原始数据类型

JavaScript 的类型分为两种：原始数据类型（Primitive data types）和对象类型（Object types）。

原始数据类型包括：布尔值、数值、字符串、`null`、`undefined` 以及 ES6 中的新类型 `Symbol`。

本节主要介绍前五种原始数据类型在 TypeScript 中的应用。

布尔值

布尔值是最基础的数据类型，在 TypeScript 中，使用 `boolean` 定义布尔值类型：

```
let isDone: boolean = false;

// 编译通过
// 后面约定，未强调编译错误的代码片段，默认为编译通过
```

注意，使用构造函数 `Boolean` 创造的对象不是布尔值：

```
let createdByNewBoolean: boolean = new Boolean(1);

// index.ts(1,5): error TS2322: Type 'Boolean' is not assignable
 to type 'boolean'.
// 后面约定，注释中标出了编译报错的代码片段，表示编译未通过
```

事实上 `new Boolean()` 返回的是一个 `Boolean` 对象：

```
let createdByNewBoolean: Boolean = new Boolean(1);
```

直接调用 `Boolean` 也可以返回一个 `boolean` 类型：

```
let createdByBoolean: boolean = Boolean(1);
```

在 TypeScript 中，`boolean` 是 JavaScript 中的基本类型，而 `Boolean` 是 JavaScript 中的构造函数。其他基本类型（除了 `null` 和 `undefined` ）一样，不再赘述。

数值

使用 `number` 定义数值类型：

```
let decLiteral: number = 6;
let hexLiteral: number = 0xf00d;
// ES6 中的二进制表示法
let binaryLiteral: number = 0b1010;
// ES6 中的八进制表示法
let octalLiteral: number = 0o744;
let notANumber: number = NaN;
let infinityNumber: number = Infinity;
```

编译结果：

```
var decLiteral = 6;
var hexLiteral = 0xf00d;
// ES6 中的二进制表示法
var binaryLiteral = 10;
// ES6 中的八进制表示法
var octalLiteral = 484;
var notANumber = NaN;
var infinityNumber = Infinity;
```

其中 `0b1010` 和 `0o744` 是 ES6 中的二进制和八进制表示法，它们会被编译为十进制数字。

字符串

使用 `string` 定义字符串类型：

```
let myName: string = 'Tom';
let myAge: number = 25;

// 模板字符串
let sentence: string = `Hello, my name is ${myName}.
I'll be ${myAge + 1} years old next month.`;
```

编译结果：

```
var myName = 'Tom';
var myAge = 25;
// 模板字符串
var sentence = "Hello, my name is " + myName + ".\nI'll be " + (
myAge + 1) + " years old next month.";
```

其中 `` ` `` 用来定义 ES6 中的模板字符串，`${expr}` 用来在模板字符串中嵌入表达式。

空值

JavaScript 没有空值（Void）的概念，在 TypeScirpt 中，可以用 `void` 表示没有任何返回值的函数：

```
function alertName(): void {
    alert('My name is Tom');
}
```

声明一个 `void` 类型的变量没有什么用，因为你只能将它赋值为 `undefined` 和 `null`：

```
let unusable: void = undefined;
```

Null 和 Undefined

在 TypeScript 中，可以使用 `null` 和 `undefined` 来定义这两个原始数据类型：

```
let u: undefined = undefined;
let n: null = null;
```

`undefined` 类型的变量只能被赋值为 `undefined`，`null` 类型的变量只能被赋值为 `null`。

与 `void` 的区别是，`undefined` 和 `null` 是所有类型的子类型。也就是说 `undefined` 类型的变量，可以赋值给 `number` 类型的变量：

```
// 这样不会报错
let num: number = undefined;
```

```
// 这样也不会报错
let u: undefined;
let num: number = u;
```

而 `void` 类型的变量不能赋值给 `number` 类型的变量：

```
let u: void;
let num: number = u;

// index.ts(2,5): error TS2322: Type 'void' is not assignable to
 type 'number'.
```

参考

- Basic Types（中文版）
- Primitive data types
- ES6 中的新类型 `Symbol`
- ES6 中的二进制和八进制表示法
- ES6 中的模板字符串

- 上一章：基础
- 下一章：任意值

任意值

任意值（Any）用来表示允许赋值为任意类型。

什么是任意值类型

如果是一个普通类型，在赋值过程中改变类型是不被允许的：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable
to type 'string'.
```

但如果是 `any` 类型，则允许被赋值为任意类型。

```
let myFavoriteNumber: any = 'seven';
myFavoriteNumber = 7;
```

任意值的属性和方法

在任意值上访问任何属性都是允许的：

```
let anyThing: any = 'hello';
console.log(anyThing.myName);
console.log(anyThing.myName.firstName);
```

也允许调用任何方法：

```
let anyThing: any = 'Tom';
anyThing.setName('Jerry');
anyThing.setName('Jerry').sayHello();
anyThing.myName.setFirstName('Cat');
```

可以认为，声明一个变量为任意值之后，对它的任何操作，返回的内容的类型都是任意值。

未声明类型的变量

变量如果在声明的时候，未指定其类型，那么它会被识别为任意值类型：

```
let something;
something = 'seven';
something = 7;

something.setName('Tom');
```

等价于

```
let something: any;
something = 'seven';
something = 7;

something.setName('Tom');
```

参考

- Basic Types # Any（中文版）

- 上一章：原始数据类型
- 下一章：类型推论

类型推论

如果没有明确的指定类型，那么 TypeScript 会依照类型推论（Type Inference）的规则推断出一个类型。

什么是类型推论

以下代码虽然没有指定类型，但是会在编译的时候报错：

```
let myFavoriteNumber = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable
to type 'string'.
```

事实上，它等价于：

```
let myFavoriteNumber: string = 'seven';
myFavoriteNumber = 7;

// index.ts(2,1): error TS2322: Type 'number' is not assignable
to type 'string'.
```

TypeScript 会在没有明确的指定类型的时候推测出一个类型，这就是类型推论。

如果定义的时候没有赋值，不管之后有没有赋值，都会被推断成 `any` 类型而完全不被类型检查：

```
let myFavoriteNumber;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

参考

- Type Inference（中文版）

- 上一章：任意值
- 下一章：联合类型

联合类型

联合类型（Union Types）表示取值可以为多种类型中的一种。

简单的例子

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
myFavoriteNumber = 7;
```

```
let myFavoriteNumber: string | number;
myFavoriteNumber = true;

// index.ts(2,1): error TS2322: Type 'boolean' is not assignable
 to type 'string | number'.
//   Type 'boolean' is not assignable to type 'number'.
```

联合类型使用 `|` 分隔每个类型。

这里的 `let myFavoriteNumber: string | number` 的含义是，允许 `myFavoriteNumber` 的类型是 `string` 或者 `number`，但是不能是其他类型。

访问联合类型的属性或方法

当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法：

```
function getLength(something: string | number): number {
    return something.length;
}

// index.ts(2,22): error TS2339: Property 'length' does not exis
t on type 'string | number'.
//   Property 'length' does not exist on type 'number'.
```

上例中，`length` 不是 `string` 和 `number` 的共有属性，所以会报错。

访问 `string` 和 `number` 的共有属性是没问题的：

```
function getString(something: string | number): string {
    return something.toString();
}
```

联合类型的变量在被赋值的时候，会根据类型推论的规则推断出一个类型：

```
let myFavoriteNumber: string | number;
myFavoriteNumber = 'seven';
console.log(myFavoriteNumber.length); // 5
myFavoriteNumber = 7;
console.log(myFavoriteNumber.length); // 编译时报错

// index.ts(5,30): error TS2339: Property 'length' does not exis
t on type 'number'.
```

上例中，第二行的 `myFavoriteNumber` 被推断成了 `string`，访问它的 `length` 属性不会报错。

而第四行的 `myFavoriteNumber` 被推断成了 `number`，访问它的 `length` 属性时就报错了。

参考

- Advanced Types # Union Types（中文版）

- 上一章：类型推论
- 下一章：对象的类型——接口

对象的类型——接口

在 TypeScript 中，我们使用接口（Interfaces）来定义对象的类型。

什么是接口

在面向对象语言中，接口（Interfaces）是一个很重要的概念，它是对行为的抽象，而具体如何行动需要由类（classes）去实现（implements）。

TypeScript 中的接口是一个非常灵活的概念，除了可用于对类的一部分行为进行抽象以外，也常用于对「对象的形状（Shape）」进行描述。

简单的例子

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};
```

上面的例子中，我们定义了一个接口 `Person`，接着定义了一个变量 `tom`，它的类型是 `Person`。这样，我们就约束了 `tom` 的形状必须和接口 `Person` 一致。

接口一般首字母大写。有的编程语言中会建议接口的名称加上 `I` 前缀。

定义的变量比接口少了一些属性是不允许的：

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom'
};

// index.ts(6,5): error TS2322: Type '{ name: string; }' is not
assignable to type 'Person'.
//   Property 'age' is missing in type '{ name: string; }'.
```

多一些属性也是不允许的：

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// index.ts(9,5): error TS2322: Type '{ name: string; age: numbe
r; gender: string; }' is not assignable to type 'Person'.
//   Object literal may only specify known properties, and 'gend
er' does not exist in type 'Person'.
```

可见，赋值的时候，变量的形状必须和接口的形状保持一致。

可选属性

有时我们希望不要完全匹配一个形状，那么可以用可选属性：

```
interface Person {
    name: string;
    age?: number;
}

let tom: Person = {
    name: 'Tom'
};
```

```
interface Person {
    name: string;
    age?: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};
```

可选属性的含义是该属性可以不存在。

这时仍然不允许添加未定义的属性：

```
interface Person {
    name: string;
    age?: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// examples/playground/index.ts(9,5): error TS2322: Type '{ name
: string; age: number; gender: string; }' is not assignable to t
ype 'Person'.
//   Object literal may only specify known properties, and 'gend
er' does not exist in type 'Person'.
```

任意属性

有时候我们希望一个接口允许有任意的属性，可以使用如下方式：

```
interface Person {
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    name: 'Tom',
    gender: 'male'
};
```

使用 `[propName: string]` 定义了任意属性取 `string` 类型的值。

需要注意的是，一旦定义了任意属性，那么确定属性和可选属性都必须是它的子属性：

```
interface Person {
    name: string;
    age?: number;
    [propName: string]: string;
}

let tom: Person = {
    name: 'Tom',
    age: 25,
    gender: 'male'
};

// index.ts(3,5): error TS2411: Property 'age' of type 'number'
is not assignable to string index type 'string'.
// index.ts(7,5): error TS2322: Type '{ [x: string]: string | nu
mber; name: string; age: number; gender: string; }' is not assig
nable to type 'Person'.
//   Index signatures are incompatible.
//     Type 'string | number' is not assignable to type 'string'.

//       Type 'number' is not assignable to type 'string'.
```

上例中，任意属性的值允许是 `string`，但是可选属性 `age` 的值却是 `number`，`number` 不是 `string` 的子属性，所以报错了。

另外，在报错信息中可以看出，此时 `{ name: 'Tom', age: 25, gender: 'male' }` 的类型被推断成了 `{ [x: string]: string | number; name: string; age: number; gender: string; }`，这是联合类型和接口的结合。

只读属性

有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 `readonly` 定义只读属性：

```
interface Person {
    readonly id: number;
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    id: 89757,
    name: 'Tom',
    gender: 'male'
};

tom.id = 9527;

// index.ts(14,5): error TS2540: Cannot assign to 'id' because i
t is a constant or a read-only property.
```

上例中，使用 `readonly` 定义的属性 `id` 初始化后，又被赋值了，所以报错了。

注意，只读的约束存在于第一次给对象赋值的时候，而不是第一次给只读属性赋值的时候：

```
interface Person {
    readonly id: number;
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    name: 'Tom',
    gender: 'male'
};

tom.id = 89757;

// index.ts(8,5): error TS2322: Type '{ name: string; gender: string; }' is not assignable to type 'Person'.
//   Property 'id' is missing in type '{ name: string; gender: string; }'.
// index.ts(13,5): error TS2540: Cannot assign to 'id' because it is a constant or a read-only property.
```

上例中，报错信息有两处，第一处是在对 `tom` 进行赋值的时候，没有给 `id` 赋值。

第二处是在给 `tom.id` 赋值的时候，由于它是只读属性，所以报错了。

参考

- Interfaces（中文版）

- 上一章：联合类型
- 下一章：数组的类型

数组的类型

在 TypeScript 中，数组类型有多种定义方式，比较灵活。

「类型 + 方括号」表示法

最简单的方法是使用「类型 + 方括号」来表示数组：

```
let fibonacci: number[] = [1, 1, 2, 3, 5];
```

数组的项中不允许出现其他的类型：

```
let fibonacci: number[] = [1, '1', 2, 3, 5];

// index.ts(1,5): error TS2322: Type '(number | string)[]' is not assignable to type 'number[]'.
//   Type 'number | string' is not assignable to type 'number'.
//     Type 'string' is not assignable to type 'number'.
```

上例中， `[1, '1', 2, 3, 5]` 的类型被推断为 `(number | string)[]` ，这是联合类型和数组的结合。

数组的一些方法的参数也会根据数组在定义时约定的类型进行限制：

```
let fibonacci: number[] = [1, 1, 2, 3, 5];
fibonacci.push('8');

// index.ts(2,16): error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

上例中， `push` 方法只允许传入 `number` 类型的参数，但是却传了一个 `string` 类型的参数，所以报错了。

数组泛型

也可以使用数组泛型（Array Generic） `Array<elemType>` 来表示数组：

```
let fibonacci: Array<number> = [1, 1, 2, 3, 5];
```

关于泛型，可以参考泛型一章。

用接口表示数组

接口也可以用来描述数组：

```
interface NumberArray {
    [index: number]: number;
}
let fibonacci: NumberArray = [1, 1, 2, 3, 5];
```

`NumberArray` 表示：只要 `index` 的类型是 `number` ，那么值的类型必须是 `number` 。

any 在数组中的应用

一个比较常见的做法是，用 `any` 表示数组中允许出现任意类型：

```
let list: any[] = ['Xcat Liu', 25, { website: 'http://xcatliu.co
m' }];
```

类数组

类数组（Array-like Object）不是数组类型，比如 `arguments` ：

```
function sum() {
    let args: number[] = arguments;
}

// index.ts(2,7): error TS2322: Type 'IArguments' is not assignable to type 'number[]'.
//   Property 'push' is missing in type 'IArguments'.
```

事实上常见的类数组都有自己的接口定义，如 `IArguments` , `NodeList` , `HTMLCollection` 等：

```
function sum() {
    let args: IArguments = arguments;
}
```

关于内置对象，可以参考内置对象一章。

参考

- Basic Types # Array（中文版）
- Interfaces # Indexable Types（中文版）

- 上一章：对象的类型——接口
- 下一章：函数的类型

函数的类型

> 函数是 JavaScript 中的一等公民

函数声明

在 JavaScript 中，有两种常见的定义函数的方式——函数声明（Function Declaration）和函数表达式（Function Expression）：

```
// 函数声明（Function Declaration）
function sum(x, y) {
    return x + y;
}

// 函数表达式（Function Expression）
let mySum = function (x, y) {
    return x + y;
};
```

一个函数有输入和输出，要在 TypeScript 中对其进行约束，需要把输入和输出都考虑到，其中函数声明的类型定义较简单：

```
function sum(x: number, y: number): number {
    return x + y;
}
```

注意，输入多余的（或者少于要求的）参数，是不被允许的：

```
function sum(x: number, y: number): number {
    return x + y;
}
sum(1, 2, 3);

// index.ts(4,1): error TS2346: Supplied parameters do not match
 any signature of call target.
```

```
function sum(x: number, y: number): number {
    return x + y;
}
sum(1);

// index.ts(4,1): error TS2346: Supplied parameters do not match
 any signature of call target.
```

函数表达式

如果要我们现在写一个对函数表达式（Function Expression）的定义，可能会写成这样：

```
let mySum = function (x: number, y: number): number {
    return x + y;
};
```

这是可以通过编译的，不过事实上，上面的代码只对等号右侧的匿名函数进行了类型定义，而等号左边的 `mySum` ，是通过赋值操作进行类型推论而推断出来的。如果需要我们手动给 `mySum` 添加类型，则应该是这样：

```
let mySum: (x: number, y: number) => number = function (x: number
, y: number): number {
    return x + y;
};
```

注意不要混淆了 TypeScript 中的 `=>` 和 ES6 中的 `=>`。

在 TypeScript 的类型定义中，`=>` 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。

在 ES6 中，`=>` 叫做箭头函数，应用十分广泛，可以参考 ES6 中的箭头函数。

用接口定义函数的形状

我们也可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    return source.search(subString) !== -1;
}
```

可选参数

前面提到，输入多余的（或者少于要求的）参数，是不允许的。那么如何定义可选的参数呢？

与接口中的可选属性类似，我们用 `?` 表示可选的参数：

```
function buildName(firstName: string, lastName?: string) {
    if (lastName) {
        return firstName + ' ' + lastName;
    } else {
        return firstName;
    }
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
```

需要注意的是，可选参数必须接在必需参数后面。换句话说，可选参数后面不允许再出现必须参数了：

```
function buildName(firstName?: string, lastName: string) {
    if (firstName) {
        return firstName + ' ' + lastName;
    } else {
        return lastName;
    }
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName(undefined, 'Tom');

// index.ts(1,40): error TS1016: A required parameter cannot fol
low an optional parameter.
```

参数默认值

在 ES6 中，我们允许给函数的参数添加默认值，**TypeScript** 会将添加了默认值的参数识别为可选参数：

```
function buildName(firstName: string, lastName: string = 'Cat')
{
    return firstName + ' ' + lastName;
}
let tomcat = buildName('Tom', 'Cat');
let tom = buildName('Tom');
```

此时就不受「可选参数必须接在必需参数后面」的限制了：

```
function buildName(firstName: string = 'Tom', lastName: string)
{
    return firstName + ' ' + lastName;
}
let tomcat = buildName('Tom', 'Cat');
let cat = buildName(undefined, 'Cat');
```

> 关于默认参数，可以参考 ES6 中函数参数的默认值。

剩余参数

ES6 中，可以使用 `...rest` 的方式获取函数中的剩余参数（rest 参数）：

```
function push(array, ...items) {
    items.forEach(function(item) {
        array.push(item);
    });
}

let a = [];
push(a, 1, 2, 3);
```

事实上， `items` 是一个数组。所以我们可以用数组的类型来定义它：

```
function push(array: any[], ...items: any[]) {
    items.forEach(function(item) {
        array.push(item);
    });
}

let a = [];
push(a, 1, 2, 3);
```

注意，rest 参数只能是最后一个参数，关于 rest 参数，可以参考 ES6 中的 rest 参数。

重载

重载允许一个函数接受不同数量或类型的参数时，作出不同的处理。

比如，我们需要实现一个函数 `reverse` ，输入数字 `123` 的时候，输出反转的数字 `321` ，输入字符串 `'hello'` 的时候，输出反转的字符串 `'olleh'` 。

利用联合类型，我们可以这么实现：

```
function reverse(x: number | string): number | string {
    if (typeof x === 'number') {
        return Number(x.toString().split('').reverse().join(''))
;
    } else if (typeof x === 'string') {
        return x.split('').reverse().join('');
    }
}
```

然而这样有一个缺点，就是不能够精确的表达，输入为数字的时候，输出也应该为数字，输入为字符串的时候，输出也应该为字符串。

这时，我们可以使用重载定义多个 `reverse` 的函数类型：

```
function reverse(x: number): number;
function reverse(x: string): string;
function reverse(x: number | string): number | string {
    if (typeof x === 'number') {
        return Number(x.toString().split('').reverse().join(''))
;
    } else if (typeof x === 'string') {
        return x.split('').reverse().join('');
    }
}
```

上例中，我们重复定义了多次函数 `reverse`，前几次都是函数定义，最后一次是函数实现。在编辑器的代码提示中，可以正确的看到前两个提示。

注意，TypeScript 会优先从最前面的函数定义开始匹配，所以多个函数定义如果有包含关系，需要优先把精确的定义写在前面。

参考

- Functions（中文版）
- Functions # Function Types（中文版）
- JS 函数式编程指南
- ES6 中的箭头函数

- ES6 中函数参数的默认值
- ES6 中的 rest 参数

- 上一章：数组的类型
- 下一章：类型断言

类型断言

类型断言（Type Assertion）可以用来手动指定一个值的类型。

语法

```
<类型>值
```

或

```
值 as 类型
```

在 tsx 语法（React 的 jsx 语法的 ts 版）中必须用后一种。

例子：将一个联合类型的变量指定为一个更加具体的类型

之前提到过，当 TypeScript 不确定一个联合类型的变量到底是哪个类型的时候，我们只能访问此联合类型的所有类型里共有的属性或方法：

```
function getLength(something: string | number): number {
    return something.length;
}

// index.ts(2,22): error TS2339: Property 'length' does not exis
t on type 'string | number'.
//   Property 'length' does not exist on type 'number'.
```

而有时候，我们确实需要在还不确定类型的时候就访问其中一个类型的属性或方法，比如：

```
function getLength(something: string | number): number {
    if (something.length) {
        return something.length;
    } else {
        return something.toString().length;
    }
}

// index.ts(2,19): error TS2339: Property 'length' does not exis
t on type 'string | number'.
//   Property 'length' does not exist on type 'number'.
// index.ts(3,26): error TS2339: Property 'length' does not exis
t on type 'string | number'.
//   Property 'length' does not exist on type 'number'.
```

上例中，获取 `something.length` 的时候会报错。

此时可以使用类型断言，将 `something` 断言成 `string` ：

```
function getLength(something: string | number): number {
    if ((<string>something).length) {
        return (<string>something).length;
    } else {
        return something.toString().length;
    }
}
```

类型断言的用法如上，在需要断言的变量前加上 `<Type>` 即可。

类型断言不是类型转换，断言成一个联合类型中不存在的类型是不允许的：

```
function toBoolean(something: string | number): boolean {
    return <boolean>something;
}

// index.ts(2,10): error TS2352: Type 'string | number' cannot b
e converted to type 'boolean'.
//   Type 'number' is not comparable to type 'boolean'.
```

参考

- TypeScript Deep Dive / Type Assertion
- Advanced Types # Type Guards and Differentiating Types（中文版）

- 上一章：函数的类型
- 下一章：声明文件

声明文件

当使用第三方库时，我们需要引用它的声明文件。

声明语句

假如我们想使用第三方库，比如 jQuery，我们通常这样获取一个 `id` 是 `foo` 的元素：

```
$('#foo');
// or
jQuery('#foo');
```

但是在 TypeScript 中，我们并不知道 `$` 或 `jQuery` 是什么东西：

```
jQuery('#foo');

// index.ts(1,1): error TS2304: Cannot find name 'jQuery'.
```

这时，我们需要使用 `declare` 关键字来定义它的类型，帮助 TypeScript 判断我们传入的参数类型对不对：

```
declare var jQuery: (string) => any;

jQuery('#foo');
```

`declare` 定义的类型只会用于编译时的检查，编译结果中会被删除。

上例的编译结果是：

```
jQuery('#foo');
```

声明文件

通常我们会把类型声明放到一个单独的文件中，这就是声明文件：

```
// jQuery.d.ts

declare var jQuery: (string) => any;
```

> 我们约定声明文件以 `.d.ts` 为后缀。

然后在使用到的文件的开头，用「三斜线指令」表示引用了声明文件：

```
/// <reference path="./jQuery.d.ts" />

jQuery('#foo');
```

第三方声明文件

当然，jQuery 的声明文件不需要我们定义了，已经有人帮我们定义好了：jQuery in DefinitelyTyped。

我们可以直接下载下来使用，但是更推荐的是使用工具统一管理第三方库的声明文件。

社区已经有多种方式引入声明文件，不过 TypeScript 2.0 推荐使用 @types 来管理。

@types 的使用方式很简单，直接用 npm 安装对应的声明模块即可，以 jQuery 举例：

```
npm install @types/jquery --save-dev
```

可以在这个页面搜索你需要的声明文件。

参考

- Writing Declaration Files（中文版）
- Triple-Slash Directives（中文版）

- 上一章：类型断言
- 下一章：内置对象

内置对象

JavaScript 中有很多内置对象，它们可以直接在 TypeScript 中当做定义好了的类型。

内置对象是指根据标准在全局作用域（Global）上存在的对象。这里的标准是指 ECMAScript 和其他环境（比如 DOM）的标准。

ECMAScript 的内置对象

ECMAScript 标准提供的内置对象有：

`Boolean`、`Error`、`Date`、`RegExp` 等。

我们可以在 TypeScript 中将变量定义为这些类型：

```
let b: Boolean = new Boolean(1);
let e: Error = new Error('Error occurred');
let d: Date = new Date();
let r: RegExp = /[a-z]/;
```

更多的内置对象，可以查看 MDN 的文档。

而他们的定义文件，则在 TypeScript 核心库的定义文件中。

DOM 和 BOM 的内置对象

DOM 和 BOM 提供的内置对象有：

`Document`、`HTMLElement`、`Event`、`NodeList` 等。

TypeScript 中会经常用到这些类型：

```
let body: HTMLElement = document.body;
let allDiv: NodeList = document.querySelectorAll('div');
document.addEventListener('click', function(e: MouseEvent) {
  // Do something
});
```

它们的定义文件同样在 TypeScript 核心库的定义文件中。

TypeScript 核心库的定义文件

TypeScript 核心库的定义文件中定义了所有浏览器环境需要用到的类型，并且是预置在 TypeScript 中的。

当你在使用一些常用的方法的时候，TypeScript 实际上已经帮你做了很多类型判断的工作了，比如：

```
Math.pow(10, '2');

// index.ts(1,14): error TS2345: Argument of type 'string' is no
t assignable to parameter of type 'number'.
```

上面的例子中，`Math.pow` 必须接受两个 `number` 类型的参数。事实上 `Math.pow` 的类型定义如下：

```
interface Math {
    /**
     * Returns the value of a base expression taken to a specifi
ed power.
     * @param x The base value of the expression.
     * @param y The exponent value of the expression.
     */
    pow(x: number, y: number): number;
}
```

再举一个 DOM 中的例子：

```
document.addEventListener('click', function(e) {
    console.log(e.targetCurrent);
});

// index.ts(2,17): error TS2339: Property 'targetCurrent' does n
ot exist on type 'MouseEvent'.
```

上面的例子中，`addEventListener` 方法是在 TypeScript 核心库中定义的：

```
interface Document extends Node, GlobalEventHandlers, NodeSelect
or, DocumentEvent {
    addEventListener(type: string, listener: (ev: MouseEvent) =>
any, useCapture?: boolean): void;
}
```

所以 `e` 被推断成了 `MouseEvent`，而 `MouseEvent` 是没有 `targetCurrent` 属性的，所以报错了。

注意，TypeScript 核心库的定义中不包含 Node.js 部分。

用 TypeScript 写 Node.js

Node.js 不是内置对象的一部分，如果想用 TypeScript 写 Node.js，则需要引入第三方声明文件：

```
npm install @types/node --save-dev
```

参考

- 内置对象
- TypeScript 核心库的定义文件

- 下一章：进阶

进阶

本部分介绍一些高级的类型与技术，具体内容包括：

- 上一章：内置对象
- 下一章：类型别名

类型别名

类型别名用来给一个类型起个新名字。

简单的例子

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    }
    else {
        return n();
    }
}
```

上例中，我们使用 `type` 创建类型别名。

类型别名常用于联合类型。

参考

- Advanced Types # Type Aliases（中文版）

- 上一章：进阶
- 下一章：字符串字面量类型

字符串字面量类型

字符串字面量类型用来约束取值只能是某几个字符串中的一个。

简单的例子

```
type EventNames = 'click' | 'scroll' | 'mousemove';
function handleEvent(ele: Element, event: EventNames) {
    // do something
}

handleEvent(document.getElementById('hello'), 'scroll');  // 没问题

handleEvent(document.getElementById('world'), 'dbclick'); // 报错，event 不能为 'dbclick'

// index.ts(7,47): error TS2345: Argument of type '"dbclick"' is not assignable to parameter of type 'EventNames'.
```

上例中，我们使用 `type` 定了一个字符串字面量类型 `EventNames`，它只能取三种字符串中的一种。

注意，类型别名与字符串字面量类型都是使用 **`type`** 进行定义。

参考

- Advanced Types # Type Aliases（中文版）

- 上一章：类型别名
- 下一章：元组

元组

数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象。

元组起源于函数编程语言（如 F#）,在这些语言中频繁使用元组。

简单的例子

定义一对值分别为 `string` 和 `number` 的元组：

```
let xcatliu: [string, number] = ['Xcat Liu', 25];
```

当赋值或访问一个已知索引的元素时，会得到正确的类型：

```
let xcatliu: [string, number];
xcatliu[0] = 'Xcat Liu';
xcatliu[1] = 25;

xcatliu[0].slice(1);
xcatliu[1].toFixed(2);
```

也可以只赋值其中一项：

```
let xcatliu: [string, number];
xcatliu[0] = 'Xcat Liu';
```

但是当直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项。

```
let xcatliu: [string, number];
xcatliu = ['Xcat Liu', 25];
```

```
let xcatliu: [string, number] = ['Xcat Liu'];

// index.ts(1,5): error TS2322: Type '[string]' is not assignabl
e to type '[string, number]'.
//   Property '1' is missing in type '[string]'.
```

```
let xcatliu: [string, number];
xcatliu = ['Xcat Liu'];
xcatliu[1] = 25;

// index.ts(2,1): error TS2322: Type '[string]' is not assignabl
e to type '[string, number]'.
//   Property '1' is missing in type '[string]'.
```

越界的元素

当赋值给越界的元素时，它类型会被限制为元组中每个类型的联合类型：

```
let xcatliu: [string, number];
xcatliu = ['Xcat Liu', 25, 'http://xcatliu.com/'];
```

上面的例子中，数组的第三项满足联合类型 `string | number` 。

```
let xcatliu: [string, number];
xcatliu = ['Xcat Liu', 25];
xcatliu.push('http://xcatliu.com/');
xcatliu.push(true);

// index.ts(4,14): error TS2345: Argument of type 'boolean' is n
ot assignable to parameter of type 'string | number'.
//   Type 'boolean' is not assignable to type 'number'.
```

当访问一个越界的元素，也会识别为元组中每个类型的联合类型：

```
let xcatliu: [string, number];
xcatliu = ['Xcat Liu', 25, 'http://xcatliu.com/'];

console.log(xcatliu[2].slice(1));

// index.ts(4,24): error TS2339: Property 'slice' does not exist
 on type 'string | number'.
```

之前提到过，如果一个值是联合类型，我们只能访问此联合类型的所有类型里共有的属性或方法。

参考

- Basic Types # Tuple（中文版）

- 上一章：字符串字面量类型
- 下一章：枚举

枚举

枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等。

简单的例子

枚举使用 `enum` 关键字来定义：

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

枚举成员会被赋值为从 `0` 开始递增的数字，同时也会对枚举值到枚举名进行反向映射：

```
enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 0); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true

console.log(Days[0] === "Sun"); // true
console.log(Days[1] === "Mon"); // true
console.log(Days[2] === "Tue"); // true
console.log(Days[6] === "Sat"); // true
```

事实上，上面的例子会被编译为：

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 0] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));
```

手动赋值

我们也可以给枚举项手动赋值：

```
enum Days {Sun = 7, Mon = 1, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 7); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true
```

上面的例子中，未手动赋值的枚举项会接着上一个枚举项递增。

如果未手动赋值的枚举项与手动赋值的重复了，TypeScript 是不会察觉到这一点的：

```
enum Days {Sun = 3, Mon = 1, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 3); // true
console.log(Days["Wed"] === 3); // true
console.log(Days[3] === "Sun"); // false
console.log(Days[3] === "Wed"); // true
```

上面的例子中，递增到 `3` 的时候与前面的 `Sun` 的取值重复了，但是 TypeScript 并没有报错，导致 `Days[3]` 的值先是 `"Sun"` ，而后又被 `"Wed"` 覆盖了。编译的结果是：

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 3] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));
```

所以使用的时候需要注意，最好不要出现这种覆盖的情况。

手动赋值的枚举项可以不是数字，此时需要使用类型断言来让tsc无视类型检查 (编译出的js仍然是可用的)：

```
enum Days {Sun = 7, Mon, Tue, Wed, Thu, Fri, Sat = <any>"S"};
```

```
var Days;
(function (Days) {
    Days[Days["Sun"] = 7] = "Sun";
    Days[Days["Mon"] = 8] = "Mon";
    Days[Days["Tue"] = 9] = "Tue";
    Days[Days["Wed"] = 10] = "Wed";
    Days[Days["Thu"] = 11] = "Thu";
    Days[Days["Fri"] = 12] = "Fri";
    Days[Days["Sat"] = "S"] = "Sat";
})(Days || (Days = {}));
```

当然，手动赋值的枚举项也可以为小数或负数，此时后续未手动赋值的项的递增步长仍为 `1` ：

```
enum Days {Sun = 7, Mon = 1.5, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 7); // true
console.log(Days["Mon"] === 1.5); // true
console.log(Days["Tue"] === 2.5); // true
console.log(Days["Sat"] === 6.5); // true
```

常数项和计算所得项

枚举项有两种类型：常数项（constant member）和计算所得项（computed member）。

前面我们所举的例子都是常数项，一个典型的计算所得项的例子：

```
enum Color {Red, Green, Blue = "blue".length};
```

上面的例子中， `"blue".length` 就是一个计算所得项。

上面的例子不会报错，但是如果紧接在计算所得项后面的是未手动赋值的项，那么它就会因为无法获得初始值而报错：

```
enum Color {Red = "red".length, Green, Blue};

// index.ts(1,33): error TS1061: Enum member must have initializ
er.
// index.ts(1,40): error TS1061: Enum member must have initializ
er.
```

下面是常数项和计算所得项的完整定义，部分引用自中文手册 - 枚举：

当满足以下条件时，枚举成员被当作是常数：

- 不具有初始化函数并且之前的枚举成员是常数。在这种情况下，当前枚举成员的值为上一个枚举成员的值加 `1` 。但第一个枚举元素是个例外。如果它没有初始化方法，那么它的初始值为 `0` 。
- 枚举成员使用常数枚举表达式初始化。常数枚举表达式是 TypeScript 表达式的子集，它可以在编译阶段求值。当一个表达式满足下面条件之一时，它就是一

个常数枚举表达式：
  - 数字字面量
  - 引用之前定义的常数枚举成员（可以是在不同的枚举类型中定义的）如果这个成员是在同一个枚举类型中定义的，可以使用非限定名来引用
  - 带括号的常数枚举表达式
  - `+`, `-`, `~` 一元运算符应用于常数枚举表达式
  - `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `>>>`, `&`, `|`, `^` 二元运算符，常数枚举表达式做为其一个操作对象。若常数枚举表达式求值后为NaN或Infinity，则会在编译阶段报错

所有其它情况的枚举成员被当作是需要计算得出的值。

常数枚举

常数枚举是使用 `const enum` 定义的枚举类型：

```
const enum Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Lef
t, Directions.Right];
```

常数枚举与普通枚举的区别是，它会在编译阶段被删除，并且不能包含计算成员。

上例的编译结果是：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* R
ight */];
```

假如包含了计算成员，则会在编译阶段报错：

```
const enum Color {Red, Green, Blue = "blue".length};

// index.ts(1,38): error TS2474: In 'const' enum declarations member initializer must be constant expression.
```

外部枚举

外部枚举（Ambient Enums）是使用 `declare enum` 定义的枚举类型：

```
declare enum Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
```

之前提到过，`declare` 定义的类型只会用于编译时的检查，编译结果中会被删除。

上例的编译结果是：

```
var directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
```

外部枚举与声明语句一样，常出现在声明文件中。

同时使用 `declare` 和 `const` 也是可以的：

```
declare const enum Directions {
    Up,
    Down,
    Left,
    Right
}

let directions = [Directions.Up, Directions.Down, Directions.Left, Directions.Right];
```

编译结果：

```
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

> TypeScript 的枚举类型的概念来源于 C#。

参考

- Enums（中文版）
- C# Enum

- 上一章：元组
- 下一章：类

类

传统方法中，JavaScript 通过构造函数实现类的概念，通过原型链实现继承。而在 ES6 中，我们终于迎来了 `class`。

TypeScript 除了实现了所有 ES6 中的类的功能以外，还添加了一些新的用法。

这一节主要介绍类的用法，下一节再介绍如何定义类的类型。

类的概念

虽然 JavaScript 中有类的概念，但是可能大多数 JavaScript 程序员并不是非常熟悉类，这里对类相关的概念做一个简单的介绍。

- 类(Class)：定义了一件事物的抽象特点，包含它的属性和方法
- 对象（Object）：类的实例，通过 `new` 生成
- 面向对象（OOP）的三大特性：封装、继承、多态
- 封装（Encapsulation）：将对数据的操作细节隐藏起来，只暴露对外的接口。外界调用端不需要（也不可能）知道细节，就能通过对外提供的接口来访问该对象，同时也保证了外界无法任意更改对象内部的数据
- 继承（Inheritance）：子类继承父类，子类除了拥有父类的所有特性外，还有一些更具体的特性
- 多态（Polymorphism）：由继承而产生了相关的不同的类，对同一个方法可以有不同的响应。比如 `Cat` 和 `Dog` 都继承自 `Animal`，但是分别实现了自己的 `eat` 方法。此时针对某一个实例，我们无需了解它是 `Cat` 还是 `Dog`，就可以直接调用 `eat` 方法，程序会自动判断出来应该如何执行 `eat`
- 存取器（getter & setter）：用以改变属性的读取和赋值行为
- 修饰符（Modifiers）：修饰符是一些关键字，用于限定成员或类型的性质。比如 `public` 表示公有属性或方法
- 抽象类（Abstract Class）：抽象类是供其他类继承的基类，抽象类不允许被实例化。抽象类中的抽象方法必须在子类中被实现
- 接口（Interfaces）：不同类之间公有的属性或方法，可以抽象成一个接口。接口可以被类实现（implements）。一个类只能继承自另一个类，但是可以实现多个接口

ES6 中类的用法

下面我们先回顾一下 ES6 中类的用法，更详细的介绍可以参考 ECMAScript 6 入门 - Class。

属性和方法

使用 `class` 定义类，使用 `constructor` 定义构造函数。

通过 `new` 生成新实例的时候，会自动调用构造函数。

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    sayHi() {
        return `My name is ${this.name}`;
    }
}

let a = new Animal('Jack');
console.log(a.sayHi()); // My name is Jack
```

类的继承

使用 `extends` 关键字实现继承，子类中使用 `super` 关键字来调用父类的构造函数和方法。

```
class Cat extends Animal {
    constructor(name) {
        super(name); // 调用父类的 constructor(name)
        console.log(this.name);
    }
    sayHi() {
        return 'Meow, ' + super.sayHi(); // 调用父类的 sayHi()
    }
}

let c = new Cat('Tom'); // Tom
console.log(c.sayHi()); // Meow, My name is Tom
```

存取器

使用 getter 和 setter 可以改变属性的赋值和读取行为：

```
class Animal {
    constructor(name) {
        this.name = name;
    }
    get name() {
        return 'Jack';
    }
    set name(value) {
        console.log('setter: ' + value);
    }
}

let a = new Animal('Kitty'); // setter: Kitty
a.name = 'Tom'; // setter: Tom
console.log(a.name); // Jack
```

静态方法

使用 `static` 修饰符修饰的方法称为静态方法，它们不需要实例化，而是直接通过类来调用：

```
class Animal {
    static isAnimal(a) {
        return a instanceof Animal;
    }
}

let a = new Animal('Jack');
Animal.isAnimal(a); // true
a.isAnimal(a); // TypeError: a.isAnimal is not a function
```

ES7 中类的用法

ES7 中有一些关于类的提案，TypeScript 也实现了它们，这里做一个简单的介绍。

实例属性

ES6 中实例的属性只能通过构造函数中的 `this.xxx` 来定义，ES7 提案中可以直接在类里面定义：

```
class Animal {
    name = 'Jack';

    constructor() {
        // ...
    }
}

let a = new Animal();
console.log(a.name); // Jack
```

静态属性

ES7 提案中，可以使用 `static` 定义一个静态属性：

```
class Animal {
    static num = 42;

    constructor() {
        // ...
    }
}

console.log(Animal.num); // 42
```

TypeScript 中类的用法

public private 和 protected

TypeScript 可以使用三种访问修饰符（Access Modifiers），分别是 `public`、`private` 和 `protected`。

- `public` 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 `public` 的
- `private` 修饰的属性或方法是私有的，不能在声明它的类的外部访问
- `protected` 修饰的属性或方法是受保护的，它和 `private` 类似，区别是它在子类中也是允许被访问的

下面举一些例子：

```
class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom';
console.log(a.name); // Tom
```

上面的例子中，`name` 被设置为了 `public`，所以直接访问实例的 `name` 属性是允许的。

很多时候，我们希望有的属性是无法直接存取的，这时候就可以用 `private` 了：

```
class Animal {
    private name;
    public constructor(name) {
        this.name = name;
    }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom';

// index.ts(9,13): error TS2341: Property 'name' is private and only accessible within class 'Animal'.
// index.ts(10,1): error TS2341: Property 'name' is private and only accessible within class 'Animal'.
```

需要注意的是，TypeScript 编译之后的代码中，并没有限制 `private` 属性在外部的可访问性。

上面的例子编译后的代码是：

```
var Animal = (function () {
    function Animal(name) {
        this.name = name;
    }
    return Animal;
}());
var a = new Animal('Jack');
console.log(a.name);
a.name = 'Tom';
```

使用 `private` 修饰的属性或方法，在子类中也是不允许访问的：

```
class Animal {
    private name;
    public constructor(name) {
        this.name = name;
    }
}

class Cat extends Animal {
    constructor(name) {
        super(name);
        console.log(this.name);
    }
}

// index.ts(11,17): error TS2341: Property 'name' is private and
 only accessible within class 'Animal'.
```

而如果是用 `protected` 修饰，则允许在子类中访问：

```
class Animal {
    protected name;
    public constructor(name) {
        this.name = name;
    }
}

class Cat extends Animal {
    constructor(name) {
        super(name);
        console.log(this.name);
    }
}
```

抽象类

`abstract` 用于定义抽象类和其中的抽象方法。

什么是抽象类？

首先，抽象类是不允许被实例化的：

```
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi();
}

let a = new Animal('Jack');

// index.ts(9,11): error TS2511: Cannot create an instance of th
e abstract class 'Animal'.
```

上面的例子中，我们定义了一个抽象类 `Animal`，并且定义了一个抽象方法 `sayHi`。在实例化抽象类的时候报错了。

其次，抽象类中的抽象方法必须被子类实现：

```
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi();
}

class Cat extends Animal {
    public eat() {
        console.log(`${this.name} is eating.`);
    }
}

let cat = new Cat('Tom');

// index.ts(9,7): error TS2515: Non-abstract class 'Cat' does no
t implement inherited abstract member 'sayHi' from class 'Animal
'.
```

上面的例子中，我们定义了一个类 `Cat` 继承了抽象类 `Animal`，但是没有实现抽象方法 `sayHi`，所以编译报错了。

下面是一个正确使用抽象类的例子：

```
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi();
}

class Cat extends Animal {
    public sayHi() {
        console.log(`Meow, My name is ${this.name}`);
    }
}

let cat = new Cat('Tom');
```

上面的例子中，我们实现了抽象方法 `sayHi`，编译通过了。

需要注意的是，即使是抽象方法，TypeScript 的编译结果中，仍然会存在这个类，上面的代码的编译结果是：

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype
= b.prototype, new __());
};
var Animal = (function () {
    function Animal(name) {
        this.name = name;
    }
    return Animal;
}());
var Cat = (function (_super) {
    __extends(Cat, _super);
    function Cat() {
        _super.apply(this, arguments);
    }
    Cat.prototype.sayHi = function () {
        console.log('Meow, My name is ' + this.name);
    };
    return Cat;
}(Animal));
var cat = new Cat('Tom');
```

类的类型

给类加上 TypeScript 的类型很简单，与接口类似：

```
class Animal {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
    sayHi(): string {
      return `My name is ${this.name}`;
    }
}

let a: Animal = new Animal('Jack');
console.log(a.sayHi()); // My name is Jack
```

参考

- Classes（中文版）
- ECMAScript 6 入门 - Class

- 上一章：枚举
- 下一章：类与接口

类与接口

之前学习过，接口（Interfaces）可以用于对「对象的形状（Shape）」进行描述。

这一章主要介绍接口的另一个用途，对类的一部分行为进行抽象。

类实现接口

实现（implements）是面向对象中的一个重要概念。一般来讲，一个类只能继承自另一个类，有时候不同类之间可以有一些共有的特性，这时候就可以把特性提取成接口（interfaces），用 `implements` 关键字来实现。这个特性大大提高了面向对象的灵活性。

举例来说，门是一个类，防盗门是门的子类。如果防盗门有一个报警器的功能，我们可以简单的给防盗门添加一个报警方法。这时候如果有另一个类，车，也有报警器的功能，就可以考虑把报警器提取出来，作为一个接口，防盗门和车都去实现它：

```
interface Alarm {
    alert();
}

class Door {
}

class SecurityDoor extends Door implements Alarm {
    alert() {
        console.log('SecurityDoor alert');
    }
}

class Car implements Alarm {
    alert() {
        console.log('Car alert');
    }
}
```

一个类可以实现多个接口：

```
interface Alarm {
    alert();
}

interface Light {
    lightOn();
    lightOff();
}

class Car implements Alarm, Light {
    alert() {
        console.log('Car alert');
    }
    lightOn() {
        console.log('Car light on');
    }
    lightOff() {
        console.log('Car light off');
    }
}
```

上例中，`Car` 实现了 `Alarm` 和 `Light` 接口，既能报警，也能开关车灯。

接口继承接口

接口与接口之间可以是继承关系：

```
interface Alarm {
    alert();
}

interface LightableAlarm extends Alarm {
    lightOn();
    lightOff();
}
```

上例中，我们使用 `extends` 使 `LightableAlarm` 继承 `Alarm`。

接口继承类

接口也可以继承类：

```
class Point {
    x: number;
    y: number;
}

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

混合类型

之前学习过，可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    return source.search(subString) !== -1;
}
```

有时候，一个函数还可以有自己的属性和方法：

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function (start: number) { };
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

参考

- Interfaces（中文版）

- 上一章：类
- 下一章：泛型

泛型

泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

简单的例子

首先，我们来实现一个函数 `createArray`，它可以创建一个指定长度的数组，同时将每一项都填充一个默认值：

```
function createArray(length: number, value: any): Array<any> {
    let result = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

上例中，我们使用了之前提到过的数组泛型来定义返回值的类型。

这段代码编译不会报错，但是一个显而易见的缺陷是，它并没有准确的定义返回值的类型：

`Array<any>` 允许数组的每一项都为任意类型。但是我们预期的是，数组中每一项都应该是输入的 `value` 的类型。

这时候，泛型就派上用场了：

```
function createArray<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray<string>(3, 'x'); // ['x', 'x', 'x']
```

上例中，我们在函数名后添加了 `<T>` ，其中 `T` 用来指代任意输入的类型，在后面的输入 `value: T` 和输出 `Array<T>` 中即可使用了。

接着在调用的时候，可以指定它具体的类型为 `string` 。当然，也可以不手动指定，而让类型推论自动推算出来：

```
function createArray<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

多个类型参数

定义泛型的时候，可以一次定义多个类型参数：

```
function swap<T, U>(tuple: [T, U]): [U, T] {
    return [tuple[1], tuple[0]];
}

swap([7, 'seven']); // ['seven', 7]
```

上例中，我们定义了一个 `swap` 函数，用来交换输入的元组。

泛型约束

在函数内部使用泛型变量的时候，由于事先不知道它是哪种类型，所以不能随意的操作它的属性或方法：

```
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length);
    return arg;
}

// index.ts(2,19): error TS2339: Property 'length' does not exis
t on type 'T'.
```

上例中，泛型 `T` 不一定包含属性 `length`，所以编译的时候报错了。

这时，我们可以对泛型进行约束，只允许这个函数传入那些包含 `length` 属性的变量。这就是泛型约束：

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    return arg;
}
```

上例中，我们使用了 `extends` 约束了泛型 `T` 必须符合接口 `Lengthwise` 的形状，也就是必须包含 `length` 属性。

此时如果调用 `loggingIdentity` 的时候，传入的 `arg` 不包含 `length`，那么在编译阶段就会报错了：

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    return arg;
}

loggingIdentity(7);

// index.ts(10,17): error TS2345: Argument of type '7' is not as
signable to parameter of type 'Lengthwise'.
```

多个类型参数之间也可以互相约束：

```
function copyFields<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = (<T>source)[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };

copyFields(x, { b: 10, d: 20 });
```

上例中，我们使用了两个类型参数，其中要求 `T` 继承 `U`，这样就保证了 `U` 上不会出现 `T` 中不存在的字段。

泛型接口

之前学习过，可以使用接口的方式来定义一个函数需要符合的形状：

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    return source.search(subString) !== -1;
}
```

当然也可以使用含有泛型的接口来定义函数的形状：

```
interface CreateArrayFunc {
    <T>(length: number, value: T): Array<T>;
}

let createArray: CreateArrayFunc;
createArray = function<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

进一步，我们可以把泛型参数提前到接口名上：

```
interface CreateArrayFunc<T> {
    (length: number, value: T): Array<T>;
}

let createArray: CreateArrayFunc<any>;
createArray = function<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']
```

注意，此时在使用泛型接口的时候，需要定义泛型的类型。

泛型类

与泛型接口类似，泛型也可以用于类的类型定义中：

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

泛型参数的默认类型

在 TypeScript 2.3 以后，我们可以为泛型中的类型参数指定默认类型。当使用泛型时没有在代码中直接指定类型参数，从实际值参数中也无法推测出时，这个默认类型就会起作用。

```
function createArray<T = string>(length: number, value: T): Array
<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}
```

参考

- Generics（中文版）
- Generic parameter defaults

- 上一章：类与接口
- 下一章：声明合并

声明合并

如果定义了两个相同名字的函数、接口或类，那么它们会合并成一个类型：

函数的合并

之前学习过，我们可以使用重载定义多个函数类型：

```
function reverse(x: number): number;
function reverse(x: string): string;
function reverse(x: number | string): number | string {
    if (typeof x === 'number') {
        return Number(x.toString().split('').reverse().join(''));
    } else if (typeof x === 'string') {
        return x.split('').reverse().join('');
    }
}
```

接口的合并

接口中的属性在合并时会简单的合并到一个接口中：

```
interface Alarm {
    price: number;
}
interface Alarm {
    weight: number;
}
```

相当于：

```
interface Alarm {
    price: number;
    weight: number;
}
```

注意，合并的属性的类型必须是唯一的：

```
interface Alarm {
    price: number;
}
interface Alarm {
    price: number;  // 虽然重复了，但是类型都是 `number`，所以不会报错
    weight: number;
}
```

```
interface Alarm {
    price: number;
}
interface Alarm {
    price: string;  // 类型不一致，会报错
    weight: number;
}

// index.ts(5,3): error TS2403: Subsequent variable declarations must have the same type.  Variable 'price' must be of type 'number', but here has type 'string'.
```

接口中方法的合并，与函数的合并一样：

```
interface Alarm {
    price: number;
    alert(s: string): string;
}
interface Alarm {
    weight: number;
    alert(s: string, n: number): string;
}
```

相当于：

```
interface Alarm {
    price: number;
    weight: number;
    alert(s: string): string;
    alert(s: string, n: number): string;
}
```

类的合并

类的合并与接口的合并规则一致。

参考

- Declaration Merging（中文版）

- 上一章：泛型
- 下一章：扩展阅读

扩展阅读

此处记录了官方手册（中文版）中包含，但是本书未涉及的概念。

我认为它们是一些不重要或者不属于 TypeScript 的概念，所以这里只给出一个简单的释义，详细内容可以点击链接深入理解。

- Never（中文版）：永远不存在值的类型，一般用于错误处理函数
- Variable Declarations（中文版）：使用 `let` 和 `const` 替代 `var`，这是 ES6 的知识
- `this`：箭头函数的运用，这是 ES6 的知识
- Using Class Types in Generics（中文版）：创建工厂函数时，需要引用构造函数的类类型
- Best common type（中文版）：数组的类型推论
- Contextual Type（中文版）：函数输入的类型推论
- Type Compatibility（中文版）：允许不严格符合类型，只需要在一定规则下兼容即可
- Advanced Types（中文版）：使用 `&` 将多种类型的共有部分叠加成一种类型
- Type Guards and Differentiating Types（中文版）：联合类型在一些情况下被识别为特定的类型
- Discriminated Unions（中文版）：使用 `|` 联合多个接口的时候，通过一个共有的属性形成可辨识联合
- Polymorphic `this` types（中文版）：父类的某个方法返回 `this`，当子类继承父类后，子类的实例调用此方法，返回的 `this` 能够被 TypeScript 正确的识别为子类的实例。
- Symbols（中文版）：新原生类型，这是 ES6 的知识
- Iterators and Generators（中文版）：迭代器，这是 ES6 的知识
- Namespaces（中文版）：避免全局污染，现在已被 ES6 Module 替代
- Decorators（中文版）：修饰器，这是 ES7 的一个提案
- Mixins（中文版）：一种编程模式，与 TypeScript 没有直接关系，可以参考 ES6 中 Mixin 模式的实现

- 上一章：声明合并

- 下一章：工程

工程

掌握了 TypeScript 的语法就像学会了砌墙的工艺。

我们学习 TypeScript 的目的不是为了造一间小茅屋，而是为了造高楼大厦，这也正是 TypeScript 的类型系统带来的优势。

那么一项大工程应该如何开展呢？本部分的内容就会介绍 TypeScript 工程化的最佳实践，具体内容包括：

- 代码检查

- 上一章：扩展阅读
- 下一章：代码检查

代码检查

目前 TypeScript 的代码检查主要有两个方案：使用 TSLint 或使用 ESLint + `typescript-eslint-parser`。

什么是代码检查

代码检查主要是用来发现代码错误、统一代码风格。

在 JavaScript 项目中，我们一般使用 ESLint 来进行代码检查。它通过插件化的特性极大的丰富了适用范围，搭配 `typescript-eslint-parser` 之后，甚至可以用来检查 TypeScript 代码。

TSLint 与 ESLint 类似，不过除了能检查常规的 js 代码风格之外，TSLint 还能够通过 TypeScript 的语法解析，利用类型系统做一些 ESLint 做不到的检查。

为什么需要代码检查

有人会觉得，JavaScript 非常灵活，所以需要代码检查。而 TypeScript 已经能够在编译阶段检查出很多问题了，为什么还需要代码检查呢？

因为 TypeScript 关注的重心是类型的匹配，而不是代码风格。当团队的人员越来越多时，同样的逻辑不同的人写出来可能会有很大的区别：

- 缩进应该是四个空格还是两个空格？
- 是否应该禁用 `var`？
- 接口名是否应该以 `I` 开头？
- 是否应该强制使用 `===` 而不是 `==`？

这些问题 TypeScript 不会关注，但是却影响到多人协作开发时的效率、代码的可理解性以及可维护性。

下面来看一个具体的例子：

```
let myName = 'Tom';

console.log(`My name is ${myNane}`);
console.log(`My name is ${myName.toStrng()}`);
console.log(`My name is ${myName}`)

// tsc 报错信息：
//
// index.ts(3,27): error TS2552: Cannot find name 'myNane'. Did you mean 'myName'?
// index.ts(4,34): error TS2551: Property 'toStrng' does not exist on type 'string'. Did you mean 'toString'?
//
//
//
// eslint 报错信息：
//
// /path/to/index.ts
//   3:27  error  'myNane' is not defined         no-undef
//   5:38  error  Missing semicolon               semi
//
// ✖ 2 problems (2 errors, 0 warnings)
//   1 errors, 0 warnings potentially fixable with the `--fix` option.
//
//
//
// tslint 报错信息：
//
// ERROR: /path/to/index.ts[5, 36]: Missing semicolon
```

存在的问题	`tsc` 是否报错	`eslint` 是否报错	`tslint` 是否报错
`myName` 被勿写成了 `myNane`			
`toString` 被勿写成了 `toStrng`	□		
少了一个分号			

上例中，由于 `eslint` 和 `tslint` 均无法识别 `myName` 存在哪些方法，所以对于拼写错误的 `toString` 没有检查出来。

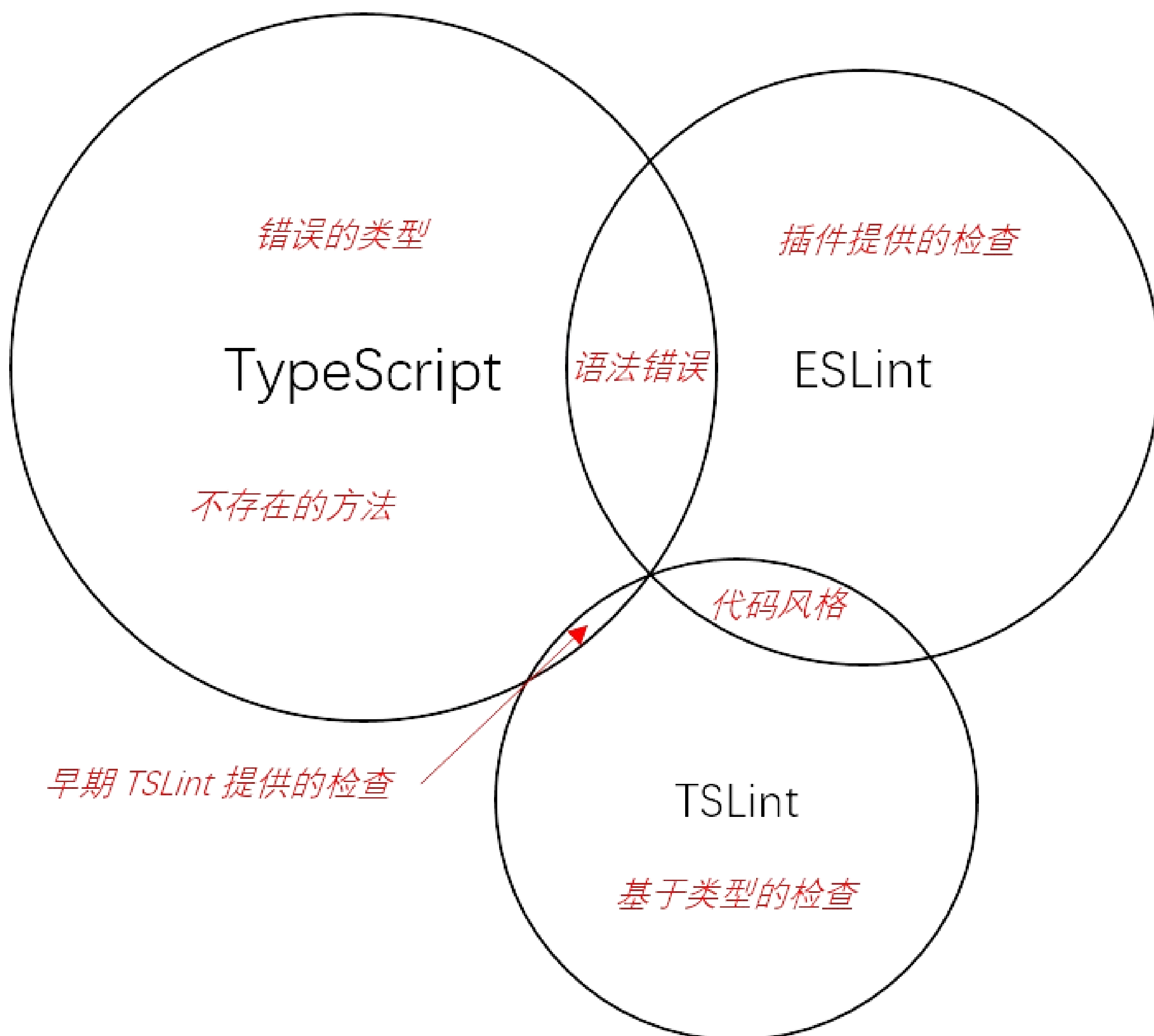
而代码风格的错误不影响编译，故少了一个分号的错误 `tsc` 没有检查出来。

对于未定义的变量 `myNane` ，`tsc` 可以检测出来。由于用到 `tslint` 的地方肯定会接入 `tsc` 编译，所以 `tslint` 就没必要检测这个错误了。`eslint` 需要能够独立于某个编译环境运行，所以能检测出此类错误，而对于 TypeScript 代码，这其实是一种冗余的检测了。

事实上，不止 `tsc` 与 `eslint` 之间有冗余的检测，`tsc` 与 `tslint` 之间也有一些冗余的检测，但是大部分都是因为早期的 `tsc` 还没能做到检测此类错误。

举个例子，TSLint 中的 `typeof-compare` 要求 `typeof` 表达式比较的对象必须是 `'undefined'`，`'object'`，`'boolean'`，`'number'`，`'string'`，`'function'` 或 `'symbol'` 之一。而 TypeScript 2.2 之后，编译器就已经自带了这个功能。

下图表示了 `tsc`，`eslint` 和 `tslint` 能覆盖的检查：



上图中， `tsc` , `eslint` 和 `tslint` 之间互相都有重叠的部分，也有各自独立的部分。

虽然发现代码错误比统一的代码风格更重要，但是当一个项目越来越庞大，开发人员也越来越多的时候，代码风格的约束还是必不可少的。

应该使用哪种代码检查工具

TSLint 与 ESLint 作为检查 TypeScript 代码的工具，各自有各自的优点：

TSLint 的优点：

1. 专为 TypeScript 服务，bug 比 ESLint 少
2. 不受限于 ESLint 使用的语法树 ESTree
3. 能直接通过 `tsconfig.json` 中的配置编译整个项目，使得在一个文件中的类型定义能够联动到其他文件中的代码检查

ESLint 的优点：

1. 基础规则比 TSLint 多很多（249 : 151）
2. 社区繁荣，插件众多（50+ : 9）

下面来看一些具体的例子：

```
let foo: string = 1 + '1';

// tslint 报错信息：
//
// ERROR: /path/to/index.ts[1, 19]: Operands of '+' operation mu
st either be both strings or both numbers, consider using templa
te literals
```

以上代码在 TSLint 中会报错，原因是加号两边必须同为数字或同为字符串（需要开启 `restrict-plus-operands` 规则）。

ESLint 无法知道加号两边的类型，所以对这种规则无能为力。

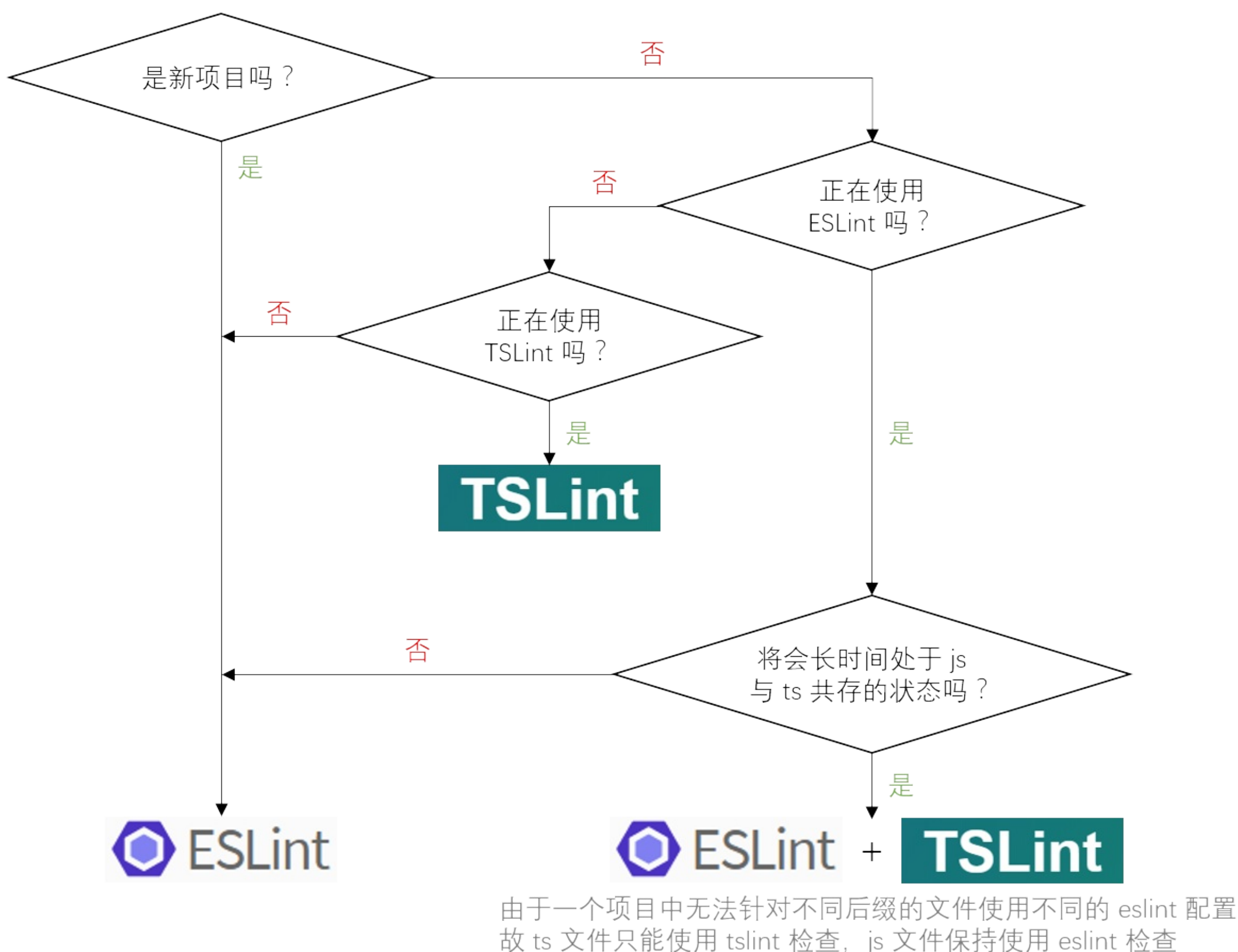
```
function foo(a, b, c, d, e, f, g, h) {
    doSomething();
}

// eslint 报错信息：
//
// /path/to/index.ts
//   1:1  error  Function 'foo' has too many parameters (8). Max
imum allowed is 7  max-params
//
// ✖ 1 problem (1 error, 0 warnings)
```

ESLint 可以检测出来以上代码的函数参数超过了 7 个（需要开启 `max-params` 规则）。

但是 TSLint 没有此项检查，虽然也可以实现，但是需要自己手动写一条规则。

那么到底该使用哪种代码检测工具呢？经过一些实践，我建议可以按照以下流程决定：



在 TypeScript 中使用 ESLint

安装 ESLint

ESLint 可以安装在当前项目中或全局环境下，因为代码检查是项目的重要组成部分，所以我们一般会将它安装在当前项目中。可以运行下面的脚本来安装：

```
npm install eslint --save-dev
```

由于 ESLint 默认使用 Espree 进行语法解析，无法识别 TypeScript 的一些语法，故我们需要安装 `typescript-eslint-parser`，替代掉默认的解析器，别忘了同时安装 `typescript`：

```
npm install typescript typescript-eslint-parser --save-dev
```

由于 `typescript-eslint-parser` 对一部分 ESLint 规则支持性不好，故我们需要安装 `eslint-plugin-typescript`，弥补一些支持性不好的规则。

```
npm install eslint-plugin-typescript --save-dev
```

创建配置文件

ESLint 需要一个配置文件来决定对哪些规则进行检查，配置文件的名称一般是 `.eslintrc.js` 或 `.eslintrc.json`。

当运行 ESLint 的时候检查一个文件的时候，它会首先尝试读取该文件的目录下的配置文件，然后再一级一级往上查找，将所找到的配置合并起来，作为当前被检查文件的配置。

我们在项目的根目录下创建一个 `.eslintrc.js`，内容如下：

```
module.exports = {
    parser: 'typescript-eslint-parser',
    plugins: [
        'typescript'
    ],
    rules: {
        // @fixable 必须使用 === 或 !==，禁止使用 == 或 !=，与 null 比较时除外
        'eqeqeq': [
            'error',
            'always',
            {
                null: 'ignore'
            }
        ],
        // 类和接口的命名必须遵守帕斯卡命名法，比如 PersianCat
        'typescript/class-name-casing': 'error'
    }
}
```

以上配置中，我们指定了两个规则，其中 `eqeqeq` 是 ESLint 原生的规则（它要求必须使用 `===` 或 `!==`，禁止使用 `==` 或 `!=`，与 `null` 比较时除外），`typescript/class-name-casing` 是 `eslint-plugin-typescript` 为 ESLint 增加的规则（它要求类和接口的命名必须遵守帕斯卡命名法，比如 `PersianCat`）。

规则的取值一般是一个数组（上例中的 `eqeqeq`），其中第一项是 `off`、`warn` 或 `error` 中的一个，表示关闭、警告和报错。后面的项都是该规则的其他配置。

如果没有其他配置的话，则可以将规则的取值简写为数组中的第一项（上例中的 `typescirpt/class-name-casing`）。

关闭、警告和报错的含义如下：

- 关闭：禁用此规则
- 警告：代码检查时输出错误信息，但是不会影响到 exit code
- 报错：发现错误时，不仅会输出错误信息，而且 exit code 将被设为 1（一般 exit code 不为 0 则表示执行出现错误）

检查一个 ts 文件

创建了配置文件之后，我们来创建一个 ts 文件看看是否能用 ESLint 去检查它了。

创建一个新文件 `index.ts`，将以下内容复制进去：

```
interface person {
    name: string;
    age: number;
}

let tom: person = {
    name: 'Tom',
    age: 25
};

if (tom.age == 25) {
    console.log(tom.name + 'is 25 years old.');
}
```

然后执行以下命令：

```
./node_modules/.bin/eslint index.ts
```

则会得到如下报错信息：

```
/path/to/index.ts
   1:11  error  Interface 'person' must be PascalCased  typescri
pt/class-name-casing
  11:13  error  Expected '===' and instead saw '=='      eqeqeq

✖ 2 problems (2 errors, 0 warnings)
```

上面的结果显示，刚刚配置的两个规则都生效了：接口 `person` 必须写成帕斯卡命名规范，`==` 必须写成 `===` 。

需要注意的是，我们使用的是 `./node_modules/.bin/eslint` ，而不是全局的 `eslint` 脚本，这是因为代码检查是项目的重要组成部分，所以我们一般会将它安装在当前项目中。

可是每次执行这么长一段脚本颇有不便，我们可以通过在 `package.json` 中添加一个 `script` 来创建一个 npm script 来简化这个步骤：

```
{
    "scripts": {
        "eslint": "eslint index.ts"
    }
}
```

这时只需执行 `npm run eslint` 即可。

检查整个项目的 ts 文件

我们的项目源文件一般是放在 `src` 目录下，所以需要将 `package.json` 中的 `eslint` 脚本改为对一个目录进行检查。由于 `eslint` 默认不会检查 `.ts` 后缀的文件，所以需要加上参数 `--ext .ts` ：

```
{
    "scripts": {
        "eslint": "eslint src --ext .ts"
    }
}
```

此时执行 `npm run eslint` 即会检查 `src` 目录下的所有 `.ts` 后缀的文件。

在 VSCode 中集成 ESLint 检查

在编辑器中集成 ESLint 检查，可以在开发过程中就发现错误，极大的增加了开发效率。

要在 VSCode 中集成 ESLint 检查，我们需要先安装 ESLint 插件，点击「扩展」按钮，搜索 ESLint，然后安装即可。

VSCode 中的 ESLint 插件默认是不会检查 `.ts` 后缀的，需要在「文件 => 首选项 => 设置」中，添加以下配置：

```
{
    "eslint.validate": [
        "javascript",
        "javascriptreact",
        "typescript"
    ]
}
```

这时再打开一个 `.ts` 文件，将鼠标移到红色提示处，即可看到这样的报错信息了：

```
1  interface person {
2      name:  [eslint] Interface 'person' must be PascalCased (typescript/cla
3      age: n ss-name-casing)
4  }       interface person
5
6  let tom: person = {
7      name: 'Tom',
8      age: 25
9  };
10
11 if (tom.age == 25) {
12     console.log(tom.name + 'is 25 years old.');
13 }
```

使用 AlloyTeam 的 ESLint 配置

ESLint 原生的规则和 `eslint-plugin-typescript` 的规则太多了，而且原生的规则有一些在 TypeScript 中支持的不好，需要禁用掉。

这里我推荐使用 AlloyTeam ESLint 规则中的 TypeScript 版本，它已经为我们提供了一套完善的配置规则。

安装：

```
npm install --save-dev eslint typescript typescript-eslint-parser eslint-plugin-typescript eslint-config-alloy
```

在你的项目根目录下创建 `.eslintrc.js`，并将以下内容复制到文件中：

```
module.exports = {
    extends: [
        'eslint-config-alloy/typescript',
    ],
    globals: {
        // 这里填入你的项目需要的全局变量
        // 这里值为 false 表示这个全局变量不允许被重新赋值，比如：
        //
        // jQuery: false,
        // $: false
    },
    rules: {
        // 这里填入你的项目需要的个性化配置，比如：
        //
        // // @fixable 一个缩进必须用两个空格替代
        // 'indent': [
        //     'error',
        //     2,
        //     {
        //         SwitchCase: 1,
        //         flatTernaryExpressions: true
        //     }
        // ]
    }
};
```

使用 ESLint 检查 tsx 文件

如果需要同时支持对 tsx 文件的检查，则需要对以上步骤做一些调整：

安装 `eslint-plugin-react`

```
npm install --save-dev eslint-plugin-react
```

package.json 中的 scripts.eslint 添加 `.tsx` 后缀

```
{
    "scripts": {
        "eslint": "eslint src --ext .ts,.tsx"
    }
}
```

VSCode 的配置中新增 typescriptreact 检查

```
{
    "eslint.validate": [
        "javascript",
        "javascriptreact",
        "typescript",
        "typescriptreact"
    ]
}
```

使用 AlloyTeam ESLint 规则中的 TypeScript React 版本

AlloyTeam ESLint 规则中的 TypeScript React 版本

在 TypeScirpt 中使用 TSLint

TSLint 的使用比较简单，参考官网的步骤安装到本地即可：

```
npm install --save-dev tslint
```

创建配置文件 `tslint.json`

```
{
    "rules": {
        // 必须使用 === 或 !==，禁止使用 == 或 !=，与 null 比较时除外
        "triple-equals": [
            true,
            "allow-null-check"
        ]
    },
    "linterOptions": {
        "exclude": [
            "**/node_modules/**"
        ]
    }
}
```

为 `package.json` 添加 `tslint` 脚本

```
{
    "scripts": {
        "tslint": "tslint --project . src/**/*.ts src/**/*.tsx",
    }
}
```

其中 `--project .` 会要求 `tslint` 使用当前目录的 `tsconfig.json` 配置来获取类型信息，很多规则需要类型信息才能生效。

此时执行 `npm run tslint` 即可检查整个项目。

在 VSCode 中集成 TSLint 检查

在 VSCode 中安装 `tslint` 插件即可，安装好之后，默认是开启的状态。

使用 AlloyTeam 的 TSLint 配置

AlloyTeam 为 TSLint 也打造了一套配置 tslint-config-alloy

```
npm install --save-dev tslint-config-alloy
```

安装之后修改 `tsconfig.json` 即可

```
{
    "extends": "tslint-config-alloy",
    "rules": {
        // 这里填入你的项目需要的个性化配置，比如：
        //
        // 一个缩进必须用两个空格替代
        // "indent": [
        //     true,
        //     "spaces",
        //     2
        // ]
    },
    "linterOptions": {
        "exclude": [
            "**/node_modules/**"
        ]
    }
}
```

使用 TSLint 检查 tsx 文件

TSLint 默认支持对 tsx 文件的检查，不需要做额外配置。

Troubleshootings

Cannot find module 'typescript-eslint-parser'

你运行的是全局的 eslint，需要改为运行 `./node_modules/.bin/eslint` 。

cannot read property type of null

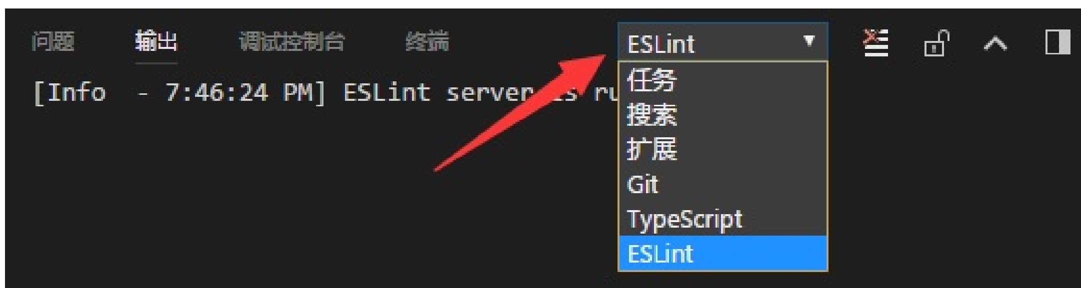
需要关闭 `eslint-plugin-react` 中的规则 `react/jsx-indent` 。

如果仍然报错，多半是因为某些规则需要被关闭，可以使用「二分排错法」检查是哪个规则造成了错误。也欢迎给 eslint-config-alloy 提 issue。

VSCode 没有显示出 ESLint 的报错

1. 检查「文件 => 首选项 => 设置」中有没有配置正确
2. 检查必要的 npm 包有没有安装
3. 检查 `.eslintrc.js` 有没有配置
4. 检查文件是不是在 `.eslintignore` 中

如果以上步骤都不奏效，则可以在「文件 => 首选项 => 设置」中配置 `"eslint.trace.server": "messages"`，按 `Ctrl` + `Shift` + `U` 打开输出面板，然后选择 ESLint 输出，查看具体错误。



为什么 ESLint 无法检查出使用了未定义的变量（`no-undef` 规则为什么被关闭了）？

因为 `typescript-eslint-parser` 无法支持 `no-undef` 规则。它针对正确的接口定义会报错。

所以我们一般会关闭 `no-undef` 规则。

为什么有些定义了的变量（比如使用 `enum` 定义的变量）未使用，ESLint 却没有报错？

因为无法支持这种变量定义的检查。建议在 `tsconfig.json` 中添加以下配置，使 `tsc` 编译过程能够检查出定义了未使用的变量：

```
{
    "compilerOptions": {
        "noUnusedLocals": true,
        "noUnusedParameters": true
    }
}
```

启用了 noUnusedParameters 之后，只使用了第二个参数，但是又必须传入第一个参数，这就会报错了

第一个参数以下划线开头即可，参考 https://github.com/Microsoft/TypeScript/issues/9458

为什么有的错误 TSLint 可以检查出来，vscode 里的 TSLint 却检查不出来？

因为 TSLint 依赖 `tsconfig.json` 获得了类型信息，而 vscode 里的 TSLint 暂不支持获取类型信息，所以 `no-unused-variable` 就失效了。

不仅 `no-unused-variables` 失效了，TSLint rules 里面所有标有 `Requires Type Info` 的规则都失效了。

- 上一章：工程
- 下一章：感谢

感谢

- 感谢创造和维护 TypeScript 的人们，给我们带来了如此优秀的工具
- 感谢 @zhongsp 对官方手册的翻译，本书参考了大量他的翻译，能一直坚持跟进非常不容易
- 感谢 @阮一峰 老师的 ECMAScript 6 入门，本书引用了多处 ES6 的知识

最后，感谢你阅读完本书，希望你会有所收获。

下一步

- 在 GitHub 上关注本书
- 阅读官方手册（中文版）巩固知识
- 阅读 Project Configuration（中文版） 学习如何配置 TypeScirpt 工程
- 查看官方示例，学习真实项目

- 上一章：代码检查